Safe and Principled Language Interoperation*

Valery Trifonov and Zhong Shao

Department of Computer Science
Yale University
New Haven, CT 06520-8285
{trifonov-valery, shao-zhong}@cs.yale.edu

Abstract. Safety of interoperation of program fragments written in different safe languages may fail when the languages have different systems of computational effects: an exception raised by an ML function may have no valid semantic interpretation in the context of a Safe-C caller. Sandboxing costs performance and still may violate the semantics if effects are not taken into account. We show that effect annotations alone are insufficient to guarantee safety, and we present a type system with bounded effect polymorphism designed to verify the compatibility of abstract resources required by the computational models of the interoperating languages. The type system ensures single address space interoperability of statically typed languages with effect mechanisms built of modules for control and state. It is shown sound for safety with respect to the semantics of a language with constructs for selection, simulation, and blocking of resources, targeted as an intermediate language for optimization of resource handling.

1 Introduction

Component-based software development promises the freedom to choose the most suitable language independently for each fragment in a system, as long as the language implementation supports a common interface [12, 15]. The existing interfaces offer a trade-off between safety and efficiency of interlanguage communication. The communicating programs may reside in separate address spaces, delegating the responsibility for correctness of their interaction to the operating system, which imposes severe performance penalties even for components using the same language implementation. Alternatively the caller and callee may share the same address space; this provides for fast interaction but typically fails to prevent possible errors due to inconsistency of the computational models. With the increasing dependence on component libraries the efficiency of the interoperation mechanism is becoming a significant factor; on the other hand the use of third-party components, especially in the context of dynamic linking, requires strong safety guarantees.

* This research was sponsored in part by the DARPA ITO under the title "Software Evolution using HOT Language Technology," DARPA Order No. D888, issued under Contract No. F30602-96-2-0232, and in part by an NSF CAREER Award CCR-9501624, and NSF Grant CCR-9633390. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED
2005		**-**

4. TITLE AND SUBTITLE	5a. CONTRACT NUMBER
Safe and Principled Language Interoperation	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **Defense Advanced Research Projects Agency,3701 North Fairfax Dr,Arlington,VA,22203-1714**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT

Safety of interoperation of program fragments written in different safe languages may fail when the languages have different systems of computational effects: an exception raised by an ML function may have no valid semantic interpretation in the context of a Safe-C caller. Sandboxing costs performance and still may violate the semantics if effects are not taken into account. We show that effect annotations alone are insufficient to guarantee safety, and we present a type system with bounded effect polymorphism designed to verify the compatibility of abstract resources required by the computational models of the interoperating languages. The type system ensures single address space interoperability of statically typed languages with effect mechanisms built of modules for control and state. It is shown sound for safety with respect to the semantics of a language with constructs for selection, simulation, and blocking of resources, targeted as an intermediate language for optimization of resource handling.

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			
unclassified	**unclassified**	**unclassified**		**19**	

Interoperation between fragments with different computational models also occurs when a higher-level language is being used for systems programming. Demands for independence of some language features (e.g. garbage collection [11]) on parts of the system can be satisfied by compiling them using a different model for a subset of the language, and using an interlanguage protocol to communicate with the rest of the system. This approach can also be used when compiling finer-grained program fragments written in the same language, to achieve pay-as-you-go efficiency of language features using the most cost-effective model which supports the features employed by the particular fragment. On a higher level, different source languages may have a common representation in a typed intermediate language [16] and different but interoperating implementation schemes, tuned to specific source language characteristics. Safety and efficiency of interoperation are again definite but competing requirements in these situations.

To provide the basis for a safe and efficient interlanguage operation, in this paper we describe a novel type-based technique, supporting principled interoperation among languages with different features selected among mutable store, exceptions, first-class continuation, and heap and stack allocation of activation records. Our framework allows programs written in multiple languages with overlapping features to interact with each other safely and reliably, yet without restricting the expressiveness of each language.

Thus our goal is designing an interoperability scheme which is

- safe: it should not be possible to violate the runtime safety of a language by calling a foreign function, even if this function is defined in a language with different features; and
- efficient: a language implementation should not be forced to use suboptimal methods for its own features in order to provide support for other languages' features. For instance the implementation of a language that does not have exceptions should not have to know about the exception handling mechanism(s) used in interoperating implementations of other languages.

Ideally we would like to have complete interoperability, allowing us to invoke any function from any term written in another language as long as the semantics of this invocation is defined in the "union" of the languages. However this requirement poses serious efficiency problems, since to satisfy it, the implementation of each language should be aware of the supporting mechanisms for all features in the union language. Thus, for instance, if a Scheme implementation S employs heap-based allocation of activation records, an implementation of Safe-C which may have to interoperate with S cannot use a stack; or, conversely, the Scheme implementations will be forced to use an allocation strategy compatible with a stack [1, 5].

At the other extreme are interfaces like COM [15] which impose few restrictions on language implementations. Safety is to be ensured via sandboxing, using separate address spaces. The interoperation mechanism supports only basic language features, e.g. function invocation and passing of arguments and result. In cases when the cost of cross-domain calls is acceptable this appears as a reasonable solution, but in fact depending on design choices in the implementations it may not provide the expected semantics.

ML:

```
exception E
fun callback () =... raise E ...
fun MLMain () =
  ...
  J.f (0,callback)
    handle E => J.f (1,callback)
  ...
```

Java:

```
class J {
  public static
    void f (int i, Callback c)
    throws Exception {
      if (i == 1) throw new Exception();
      try { c.invoke (); }
      catch (Exception e) { ... }
  }
}
```

Fig. 1. Failure of simple sandboxing to preserve semantics

Consider the schematic example shown in Figure 1, where raising exception `E` in `callback` shortcuts the flow through the Java fragment. If the Java implementation maintains information about the last entered `try` (i.e. the current exception handler) which is context-switched upon calls to ML, this information will be incorrect (with respect to the "union" semantics) when `Exception` is thrown after the second call to `J.f`.

Thus as observed in [14] the function of a mechanism for safe interlanguage calls is more than marshalling values between representations – it must also take into account the *effect systems* of the languages.

The contribution of this paper is that it formalizes the notion of safe interoperability between statically typed languages by building on previous work on effect systems [8, 18, 19] and introducing a type system which relates effects to the *machine resources* they require. Since different models of languages provide different sets of resources, and the same effect may be possible with various sets of resources, both an effect and a resource annotation are needed: the resource annotation indicates the specific requirements of a code fragment, while the effect annotation determines whether an alternative set of resources can be coerced to match these requirements.

Tracking the effects of the parameters of higher-order functions is achieved by effect polymorphism: using effect variables in the types to express the dependencies. More specifically our system has *bounded effect polymorphism* to reject effect applications when the effect arguments are unsupported by the resource bounds. We omit type polymorphism from the present description for brevity; we believe it is largely orthogonal to the treatment of resources and effects in types.

Furthermore, to show soundness of our type system, we introduce a typed language with constructs for explicit management of machine resources. Using this language as intermediate in compiling various source languages allows the compiler to optimize interlanguage calls by "floating" the boundary between contexts with different resource requirements. Thus parts of a program written in one language can be specialized to operate with the resources provided by the implementation of another language [17].

The result is that our system avoids the safety traps and allows for the interoperation of efficient implementations by restricting, in some cases, which foreign functions may be invoked, and imposing conditions the caller must satisfy before the invocation. To determine whether a call is possible, we consider the effects of a function, i.e. its use of

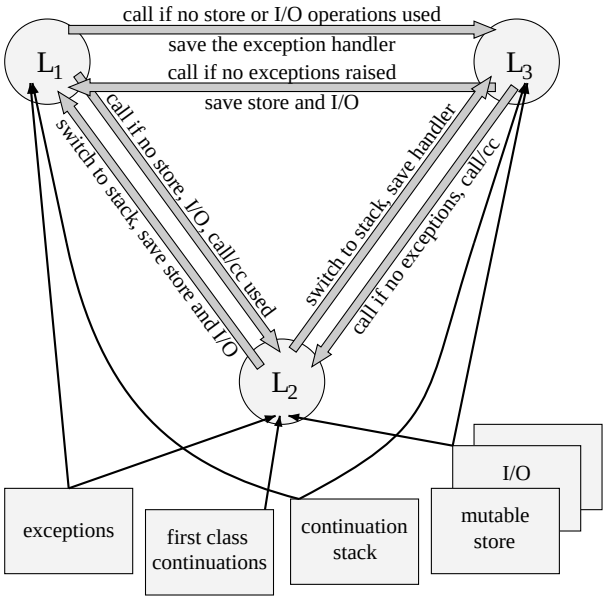


Fig. 2. Language interoperation: conditions for safety and efficiency

resources. A resource has to be saved (blocked) when it is not required by the callee. If a resource which the caller does not have is needed by the called function, but the latter does not produce an effect depending on this resource, a dummy resource can be provided instead. In some cases it is possible to switch to alternative resources supporting the same effect. An example illustrating these points is shown in Figure 2 where three languages L_1, L_2, and L_3 are built out of the semantic modules studied in this paper. For example, calling an L_3 function from an L_2 program is always possible, because the functionality of all resources of L_3 is supported in L_2, but the calling convention must be switched from heap to stack based, and the L_2 exception handler must be preserved.

To specify formally and prove the safety of our system, in Section 2 we introduce the typed intermediate language $\mathcal{R}$ and describe its static and dynamic semantics in Section 3. In Section 3.3 we show that the type system of $\mathcal{R}$ guarantees the runtime safety of type-correct programs. This makes possible the safe linking of separately compiled components which can be shown to have a type-correct translation into $\mathcal{R}$ with the corresponding interface type.

2 A Language with Machine Resource Control

The language $\mathcal{R}$ is an idealized version of the typed intermediate language of our system. The novel feature in it are the *abstract resources*, which together with their associated primitive operations can be seen as modules of which we can build sublanguages of $\mathcal{R}$; indeed this is an implied goal of our interoperability scheme. Both the static and the dynamic semantics of $\mathcal{R}$ permit a presentation in which new functional blocks are

ABSTRACT RESOURCES

– CONTINUATION

$ContRes \ni a ::= \mathsf{S} \mid \mathsf{H}$ — continuation stack and heap

– CONTROL

$CtrlRes \ni c ::= a \mid \mathsf{X}$ — … plus an exception handler

– PRIMITIVE

$PrimRes \ni r ::= c \mid \mathsf{M}$ — … plus a mutable store

RESOURCE DESCRIPTORS

$$\rho \in ResourceDesc = CtrlRes \times 2^{PrimRes}$$

EFFECTS

$$u \in EffVar$$
$$PrimEff \ni f ::= \mathsf{callcc} \mid \mathsf{exception} \mid \mathsf{store}$$
$$\varepsilon \in Effects = 2^{EffVar \cup PrimEff}$$

TYPES

$$Typ \ni \tau ::= \tau \to^{\rho}_{\varepsilon} \tau \mid \mathsf{cont}^{\rho}[\tau] \mid \forall u \leq \rho.\, \tau \mid \mathsf{exn} \mid \mathsf{ref}[\tau] \mid \mathsf{unit} \mid b, \quad b \in BasicTyp$$

VALUES AND TERMS

$$Val \ni v ::= x \mid d \quad (x \in Var,\ d \in Const \supseteq \{*\})$$
$$\mid \lambda^{\rho} x{:}\tau.\, e \quad \text{resource-specific abstractions}$$
$$\mid \Lambda u \leq \rho.\, v \quad \text{bounded effect abstractions}$$
$$\mid x[\varepsilon] \quad \text{effect applications}$$
$$Exp \ni e ::= @\ x\ x' \quad \text{applications}$$
$$\mid \mathbf{use}\,(\rho)\, e \quad \text{resource control}$$
$$\mid \lfloor v \rfloor \quad \text{values}$$
$$\mid \mathbf{let}\ x : \tau \leftarrow e\ \mathbf{in}\ e' \quad \text{bindings}$$
$$\mid \mathbf{ref}\ x \mid\ !\,x \mid x := x' \quad \text{store}$$
$$\mid \mathbf{callcc}\ x \mid \mathbf{throw}[\tau]\ x\ x' \quad \text{continuations}$$
$$\mid e\ \mathbf{handle}\ x : \mathsf{exn}.\, e' \mid \mathbf{raise}[\tau]\ x \quad \text{exceptions}$$

Fig. 3. Syntax of $\mathcal{R}$, a language with typed resource control

added to a basic language without interference with other blocks; the exception is the exceptions block, whose semantics needs support from both first-class continuations (when present) and the resource handling itself. We take the approach of presenting all blocks in one step mainly due to space constraints.

The abstract resources, ranged over by r (see Figure 3), are structured in a hierarchy including the control resources c and their subdivision, the continuation allocation resources a. The language supports two allocation strategies for activation records: a stack-based discipline with abstract resource S, and a heap-based, with abstract resource H. An additional control resource is the exception handler X. Informally all of the control resources can be viewed as structures of frames such as activation records. A primitive resource not related to the control is the store M; the system can be directly extended with multiple versions of the store which can be controlled separately.

Primitive resources are the building blocks of the resource descriptors ρ, consisting of two components. The first component specifies the "calling convention" in use. i.e.

how values are communicated to and from a term; to keep the system simple we only consider conventions on returning the result, with a choice between a stack-allocated and a heap-allocated continuation. The second component describes the set of resources available or required for the evaluation of a term.

The counterpart of resources are the *effects* ε which are sets of primitive effects (informally caused by the corresponding primitive operations in the language) and effect variables which stand for sets of effects. In our language the primitive effects are callcc, exception, and store. A computation may only introduce effects which are provided for by the available resources, e.g. the effect exception can only occur when the exception handler resource X is available.

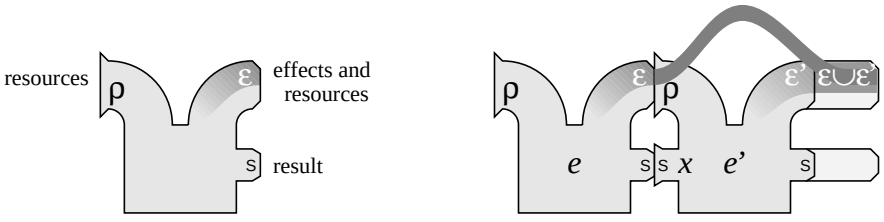
There are only minimal requirements for resources needed to produce an effect; extra resources can simply be ignored. This intuitive observation leads to the definition of a relation of compatibility between resource descriptors: letting rs range over sets of primitive resources, $\langle a, rs\rangle \sqsubseteq \langle a, rs'\rangle$ if $rs \subseteq rs'$. Note that compatible resource descriptors denote the same calling convention.

The types τ include function types $\tau_1 \to^{\rho}_{\varepsilon} \tau_2$ annotated with a resource descriptor ρ and effects ε. This notation describes a function whose evaluation requires the calling convention and resources denoted by ρ, and produces the effects ε. Similarly the resource annotation ρ on the type of continuations $\mathsf{cont}^{\rho}[\tau]$ denotes the resources needed to re-activate the continuation; the type system assumes that the effect of invoking a continuation is the maximal possible under ρ.

Among the types are also the bounded-effect quantified types $\forall u \le \rho.\,\tau$. An effect application $x[\varepsilon]$ of a variable x of this type is only valid when the effects ε are possible with the resources described in ρ.

In addition to effect applications the values v of $\mathcal{R}$ include bounded effect abstractions, and abstractions annotated with resource descriptors with the meaning noted for function types above.

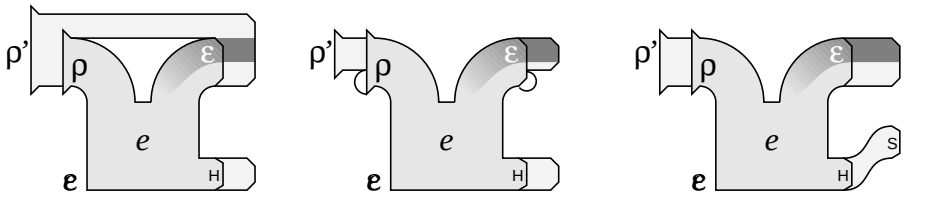
A term e which requires the resources described in ρ and produces effects ε can be visualized as the element shown in Figure 4(a), where the S indicates the convention for the result is to use a stack. The term $\lfloor v \rfloor$ denotes an effect-free computation returning the value v according to a convention determined by the context. The computation of **let** $x : \tau \leftarrow e$ **in** e' merges the effects of the computations of e and e' with x bound to the value of e (Figure 4(b)).



(a) A term requiring resources ρ and producing effects ε

(b) Evaluation of **let** $x : \tau \leftarrow e$ **in** e' with resources ρ

Fig. 4. Terms and their composition



(a) restricting (b) extending (c) switching calling conventions
the set of resources

Fig. 5. Operations on resources performed by **use** $(\rho)\ e$

A novelty is the resource-management construct **use** $(\rho)\ e$, which evaluates e in the context of resources described by ρ, replacing the current resources. For instance, it is possible to reduce the set of available resources ρ' when a term e expects a subset ρ (as reflected in its type, see Section 3 for details). As alluded to in Figure 5(a), the resources currently available but not needed are preserved during the evaluation of e and restored upon completion. For example, consider the case of an ML program with resources $\langle \mathsf{H}, \{\mathsf{H}, \mathsf{X}, \mathsf{M}\}\rangle$ invoking a Scheme function f with an integer argument x and integer result. Assuming both implementations perform heap allocation of activation records, the call would require preserving the ML exception handler. Translated to $\mathcal{R}$, the invocation is expressed by

$$\textbf{use}\ (\langle \mathsf{H}, \{\mathsf{H}, \mathsf{M}\}\rangle)\ (@\ \mathsf{f}\ \mathsf{x})$$

in a type environment including $\mathsf{x} : \mathsf{Int}, \mathsf{f} : \mathsf{Int} \to^{\langle \mathsf{H}, \{\mathsf{H},\mathsf{M}\}\rangle}_{\varepsilon} \mathsf{Int}$ (for some effect ε).

It is also useful to allow the creation of a new dummy resource r when it is required by a term e but the effects of e make no use of r. Consider a Scheme fragment with resources $\langle \mathsf{H}, \{\mathsf{H}, \mathsf{M}\}\rangle$ invoking a compiled from ML function g of type $\mathsf{Int} \to^{\langle \mathsf{H}, \{\mathsf{H},\mathsf{X},\mathsf{M}\}\rangle}_{\{\mathsf{store}\}} \mathsf{Int}$. This type shows that g has no effects that require the exception handler, but the code for g nevertheless expects an exception handler resource, which is reflected in the resource component of g's type. Therefore the invocation must be enclosed in a **use**, constructing a dummy X resource:

$$\textbf{use}\ (\langle \mathsf{H}, \{\mathsf{H}, \mathsf{X}, \mathsf{M}\rangle)\ (@\ \mathsf{g}\ 5)$$

This situation is represented graphically in Figure 5(b) where the effects ε are supported by the resources ρ', but the term e requires additional resources.

In the absence of continuation capture effects the creation of new stack and heap resources has different semantics due to the localization of allocation effects on these resources. While a newly created store or exception handler resource cannot be used at all (since any use would create an effect which requires that resource, hence the term would not have a type in an environment which does not provide the resource), a new stack or heap may be used for the allocation of activation records, because the allocation effect is localized and will not be propagated when the evaluation of the term is completed. Note that this only applies to the use of the heap for stack-like management of activation records; use of **callcc** for instance introduces an effect which

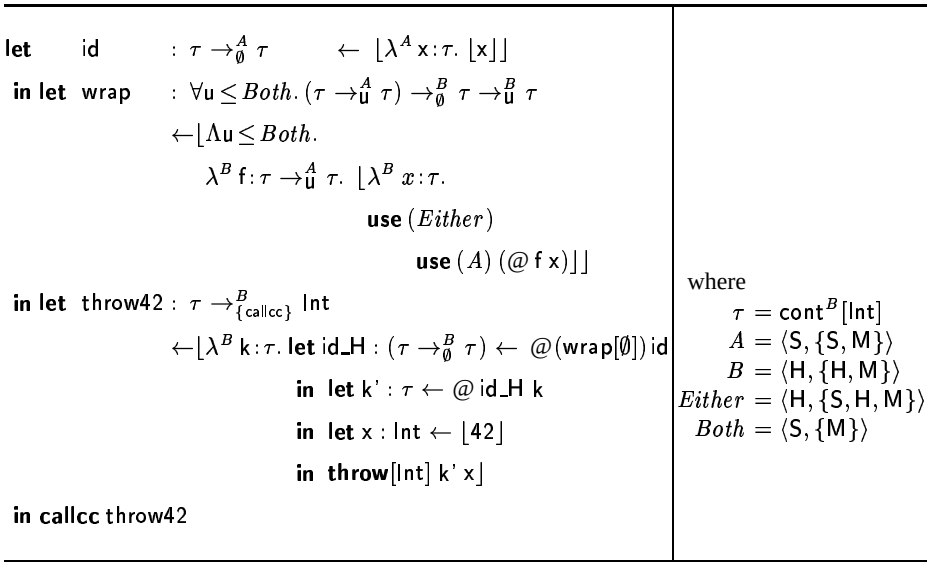


Fig. 6. Example of handling of foreign objects

makes it impossible to type the term in an environment which does not have a heap resource.

Another application of the construct **use** (ρ) e is the selection of calling convention. Figure 5(c) illustrates this in the case of switching from stack-based to heap-based continuations for the evaluation of e when $\rho = \langle \mathsf{H}, rs \rangle$ and the resource descriptor of the context is $\rho' = \langle \mathsf{S}, rs \rangle$, where $\{\mathsf{H}, \mathsf{S}\} \subseteq rs$ (i.e. both a heap and a stack are provided). Formal conditions for validity of **use** (ρ) e are presented in Section 3.

The example in Figure 6 shows an application of **use** in the case of interoperation between programs written in two languages. The function id uses stack allocation (S), while throw42 uses heap allocation (H). Both languages also have the store resource M. Before id can be called from throw42, it must be coerced to heap allocation, which is performed by the effect-polymorphic function wrap. Note that the effects of the arguments of wrap are restricted by the resources in the intersection of the two languages' sets, denoted by $Both$; in this case this means only store effects are allowed. The invocation of wrap's argument f is enclosed in two **use** regions: the outer **use** creates a new stack, while the inner one switches the calling convention to the stack and saves the heap resource.

The example also shows how an object which only has meaning in a language with a given feature can be handled "passively" by code written in a language without support for that feature. Note that the first-class continuation captured in the heap-allocating program is passed to id and back, and then activated.

EFFECT ENVIRONMENT FORMATION

(Env-eff-empty)
$$\vdash_\Delta \emptyset$$

(Env-eff-ext)
$$\frac{\vdash_\Delta \Delta}{\vdash_\Delta \Delta_t, t \leq \rho}$$

EFFECTS

(Eff-empty)
$$\frac{\vdash_\Delta \Delta}{\Delta \vdash_\varepsilon \emptyset \leq \rho}$$

(Eff-union)
$$\frac{\Delta \vdash_\varepsilon \varepsilon' \leq \rho \quad \Delta \vdash_\varepsilon \varepsilon'' \leq \rho}{\Delta \vdash_\varepsilon \varepsilon' \cup \varepsilon'' \leq \rho}$$

(Eff-var)
$$\frac{\vdash_\Delta \Delta \quad u \in Dom(\Delta)}{\Delta \vdash_\varepsilon u \leq \Delta(u)}$$

(Eff-primitive)
$$\frac{\vdash_\Delta \Delta \quad \rho \in Required(f)}{\Delta \vdash_\varepsilon \{f\} \leq \rho}$$

(Eff-add-resource)
$$\frac{\Delta \vdash_\varepsilon \varepsilon \leq \rho \quad \rho \sqsubseteq \rho'}{\Delta \vdash_\varepsilon \varepsilon \leq \rho'}$$

TYPES

(Typ-basic)
$$\frac{\vdash_\Delta \Delta}{\Delta \vdash_\tau b}$$

(Typ-fun)
$$\frac{\Delta \vdash_\varepsilon \varepsilon \leq \rho \quad \Delta \vdash_\tau \tau, \tau'}{\Delta \vdash_\tau \tau \to^\rho_\varepsilon \tau'}$$

(Typ-unit)
$$\frac{\vdash_\Delta \Delta}{\Delta \vdash_\tau \mathsf{unit}}$$

(Typ-poly)
$$\frac{\vdash_\Delta \Delta \quad \Delta_u, u \leq \rho \vdash_\tau \tau}{\Delta \vdash_\tau \forall u \leq \rho.\, \tau}$$

(Typ-ref)
$$\frac{\Delta \vdash_\tau \tau}{\Delta \vdash_\tau \mathsf{ref}[\tau]}$$

(Typ-cont)
$$\frac{\Delta \vdash_\tau \tau \quad \emptyset \vdash_\varepsilon \{\mathsf{callcc}\} \leq \rho}{\Delta \vdash_\tau \mathsf{cont}^\rho[\tau]}$$

TYPE ENVIRONMENT FORMATION

(Env-typ-empty)
$$\frac{\vdash_\Delta \Delta}{\Delta \vdash_T \emptyset}$$

(Env-typ-ext)
$$\frac{\Delta \vdash_T \Gamma \quad \Delta \vdash_\tau \tau}{\Delta \vdash_T \Gamma_x, x : \tau}$$

VALUES

(Val-const)
$$\frac{\Delta \vdash_T \Gamma}{\Delta; \Gamma \vdash_v d : \theta(d)}$$

(Val-var)
$$\frac{\Delta \vdash_T \Gamma \quad x \in Dom(\Gamma)}{\Delta; \Gamma \vdash_v x : \Gamma(x)}$$

(Val-abs)
$$\frac{\Delta \vdash_T \Gamma \quad \Delta \vdash_\tau \tau \quad \rho; \Delta; \Gamma_x, x : \tau \vdash_e e : \tau'; \varepsilon}{\Delta; \Gamma \vdash_v \lambda^\rho x{:}\tau.\, e : \tau \to^\rho_\varepsilon \tau'}$$

(Val-eff-abs)
$$\frac{\Delta \vdash_T \Gamma \quad \Delta_u, u \leq \rho; \Gamma \vdash_v v : \tau}{\Delta; \Gamma \vdash_v \Lambda u \leq \rho.\, v : \forall u \leq \rho.\, \tau}$$

(Val-eff-app)
$$\frac{\Gamma(x) = \forall u \leq \rho.\, \tau \quad \Delta \vdash_\varepsilon \varepsilon \leq \rho}{\Delta; \Gamma \vdash_v x[\varepsilon] : [\varepsilon/u]\tau}$$

Fig. 7. The $\mathcal{R}$ type system: effects, types and values

3 Semantics of $\mathcal{R}$

3.1 Static Semantics

The type system of $\mathcal{R}$, shown in Figures 7 and 8, keeps track of the resources necessary for the evaluation of a term and makes a conservative estimate of the effects of the evaluation. The effect environment Δ specifies the resource bounds of free effect variables, and as usual the type environment Γ assigns types to free variables.

The rules for sequents $\Delta \vdash_\varepsilon \varepsilon \leq \rho$ reflect the dependence of effects on resources and form the basis of bounded effect polymorphism. The dependence of primitive effects on resources is captured by the function *Required* (Figure 9) specifying the alternatives for minimal resource descriptors enabling a primitive effect. Note that the exception and store effects work with either stack or heap continuation allocation, while the callcc effect can only be introduced with heap allocation.

(Exp-app)
$$\frac{\Delta \vdash_T \Gamma \quad \Gamma(x) = \tau' \to^{\rho}_{\varepsilon} \tau \quad \Gamma(x') = \tau'}{\rho; \Delta; \Gamma \vdash_e @\ x\ x' : \tau; \varepsilon}$$

(Exp-use)
$$\frac{\rho; \Delta; \Gamma \vdash_e e : \tau; \varepsilon \quad \Delta \vdash_\varepsilon \varepsilon \leq \rho'}{\rho'; \Delta; \Gamma \vdash_e \mathbf{use}\,(\rho)\, e : \tau; \varepsilon}$$

(Exp-val)
$$\frac{\Delta; \Gamma \vdash_v v : \tau}{\rho; \Delta; \Gamma \vdash_e \lfloor v \rfloor : \tau; \emptyset}$$

(Exp-let)
$$\frac{\rho; \Delta; \Gamma \vdash_e e : \tau; \varepsilon \quad \rho; \Delta; \Gamma_x, x : \tau \vdash_e e' : \tau'; \varepsilon'}{\rho; \Delta; \Gamma \vdash_e \mathbf{let}\ x : \tau \leftarrow e\ \mathbf{in}\ e' : \tau'; \varepsilon \cup \varepsilon'}$$

(Exp-callcc)
$$\frac{\Delta \vdash_T \Gamma \quad \Gamma(x) = \mathsf{cont}^{\rho}[\tau] \to^{\rho}_{\varepsilon} \tau}{\rho; \Delta; \Gamma \vdash_e \mathbf{callcc}\ x : \tau; \varepsilon \cup \{\mathsf{callcc}\}}$$

(Exp-throw)
$$\frac{\Delta \vdash_T \Gamma \quad \Delta \vdash_\tau \tau \quad \Gamma(x) = \mathsf{cont}^{\rho}[\tau'] \quad \Gamma(x') = \tau'}{\rho; \Delta; \Gamma \vdash_e \mathbf{throw}[\tau]\ x\ x' : \tau;\ \mathit{MaxEff}\ (\rho)}$$
where $\mathit{MaxEff}\ (\rho) = \{f \mid \emptyset \vdash_\varepsilon \{f\} \leq \rho\}$

(Exp-ref)
$$\frac{\Delta \vdash_T \Gamma \quad \Gamma(x) = \tau \quad \Delta \vdash_\varepsilon \{\mathsf{store}\} \leq \rho}{\rho; \Delta; \Gamma \vdash_e \mathbf{ref}\ x : \mathsf{ref}[\tau]; \{\mathsf{store}\}}$$

(Exp-deref)
$$\frac{\Delta \vdash_T \Gamma \quad \Gamma(x) = \mathsf{ref}[\tau] \quad \Delta \vdash_\varepsilon \{\mathsf{store}\} \leq \rho}{\rho; \Delta; \Gamma \vdash_e\ !\ x : \tau; \{\mathsf{store}\}}$$

(Exp-update)
$$\frac{\Delta \vdash_T \Gamma \quad \Gamma(x) = \mathsf{ref}[\tau] \quad \Gamma(x') = \tau \quad \Delta \vdash_\varepsilon \{\mathsf{store}\} \leq \rho}{\rho; \Delta; \Gamma \vdash_e x := x' : \mathsf{unit}; \{\mathsf{store}\}}$$

(Exp-raise)
$$\frac{\Delta \vdash_T \Gamma \quad \Gamma(x) = \mathsf{exn} \quad \Delta \vdash_\varepsilon \{\mathsf{exception}\} \leq \rho}{\rho; \Delta; \Gamma \vdash_e \mathbf{raise}[\tau]\ x : \tau; \{\mathsf{exception}\}}$$

(Exp-handle)
$$\frac{\rho; \Delta; \Gamma \vdash_e e : \tau; \varepsilon \quad \rho; \Delta; \Gamma_x, x : \mathsf{exn} \vdash_e e' : \tau; \varepsilon' \quad \Delta \vdash_\varepsilon \{\mathsf{exception}\} \leq \rho}{\rho; \Delta; \Gamma \vdash_e e\ \mathbf{handle}\ x : \mathsf{exn}.\ e' : \tau; \varepsilon \cup \varepsilon' \cup \{\mathsf{exception}\}}$$

Fig. 8. The $\mathcal{R}$ type system: terms

Type judgments for values associate a type τ with a value v and a pair of environments; values have no effects and therefore their computation requires no resources. Sequents for terms have the form $\rho; \Delta; \Gamma \vdash_e e : \tau; \varepsilon$, where ρ is the resource descriptor of the environment, τ is the type of e, and ε represents the effects of the evaluation of e. For the typing of constants we assume the existence of a function $\theta \in \mathit{Const} \to \mathit{BasicTyp}$, and in particular that $\theta(*) = \mathsf{unit}$.

$$\begin{aligned} \mathit{Required}\ (\mathsf{callcc}) &= \{\langle \mathsf{H}, \{\mathsf{H}\}\rangle\} \\ \mathit{Required}\ (\mathsf{exception}) &= \{\langle \mathsf{S}, \{\mathsf{S}, \mathsf{X}\}\rangle,\ \langle \mathsf{H}, \{\mathsf{H}, \mathsf{X}\}\rangle\} \\ \mathit{Required}\ (\mathsf{store}) &= \{\langle \mathsf{S}, \{\mathsf{S}, \mathsf{M}\}\rangle,\ \langle \mathsf{H}, \{\mathsf{H}, \mathsf{M}\}\rangle\} \end{aligned}$$

Fig. 9. Minimal resource requirements for primitive effects

The function $MaxEff$ yields the maximal effect possible with the resources in ρ; it is used in making a conservative approximation of the effects of a continuation given the resources available at the point of its capture (in rule (Exp-throw)).

3.2 Dynamic Semantics

To separate the static and dynamic aspects of the use of resource descriptors we consider a variant of $\mathcal{R}$ with explicit resource annotations, where the abstract syntax for terms is

$$\begin{array}{rl} e ::= & @\ x\ x' \mid \mathbf{use}^{\rho}\ (\rho')\ e \\ & \mid \lfloor v \rfloor^{\rho} \mid \mathbf{let}^{\rho}\ x : \tau \leftarrow e\ \mathbf{in}\ e' \\ & \mid \mathbf{ref}^{\rho}\ x \mid \mathbf{!}^{\rho}\ x \mid x \mathbf{:=}^{\rho} x' \\ & \mid \mathbf{callcc}^{\rho}\ x \mid \mathbf{throw}^{\rho}[\tau]\ x\ x' \\ & \mid e\ \mathbf{handle}^{\rho}\ x : \mathsf{exn}.\ e' \mid \mathbf{raise}^{\rho}[\tau]\ x \end{array}$$

The translation of type-correct $\mathcal{R}$ terms into the annotated language is straightforward since the new annotations correspond to the resource descriptors ρ on the left of $\vdash_e$ in the typing sequents for those terms.

We present operational semantics of $\mathcal{R}$ following [3] in terms of a variant of the tail-call-safe C_aEK machine. We prefer operational semantics because it allows us to show directly that reasonably efficient implementations exist. The original machine allocates an activation frame on the continuation stack when entering a **let**-binding (but not when entering a closure), and pops the frame to complete the binding. We extend the machine by including a component denoting additional machine resources, and by providing a heap-based alternative continuation allocation strategy supporting first-class continuations.

The transitions of the abstract machine are specified as a relation on machine configurations consisting of the term being evaluated, its environment, and the currently available machine resources; for the purposes of the proof of soundness of the type system we include in the configuration also the type of the evaluated term as well as an accumulator of effects. Having the annotations in the syntax makes it clear that the although the semantics of some constructs depend on what resources are available, this dependence can be resolved at compile time.

The semantic domains defining the meaning of the components of the abstract machine are listed in Figure 10. As usual the environment E maps variables to values converted to their internal representation as shown in Figure 11. Values are represented in the environment as closures with environments binding their free variables; no environment for effect variables is needed because effect instantiation is performed via substitution (Figure 11).

To assist with the proof of soundness of the type system we instrument the operational semantics to keep track of the current control resource, the accumulated effects of the evaluation, and the type of the term being evaluated. Further, the environment is extended to also record the type of each variable; type safety (Lemma 1) implies that all type annotations and tags only used for verification in the semantics may be erased without affecting the outcome of the evaluation of a correctly typed term. We use the shorthands ${}^{V}E(x)$ and ${}^{T}E(x)$ for the value and type components of $E(x)$, respectively,

MACHINE CONFIGURATION

$$Config = Exp \times Env \times CtrlRes \times MResources \times Effects \times Typ$$

ENVIRONMENT

$$MVal \ni w ::= \mathsf{Closure}\ \langle v,\ E\rangle \mid \mathsf{Cont}\ k \mid \mathsf{Loc}\ \ell \quad \text{machine values}$$

$$E \in Env = Var \to MVal \times Typ \quad \text{environment}$$

MACHINE RESOURCES

$$Rs \in MResources = \textstyle\sum_{rs \subseteq PrimRes} \prod_{r \in rs} MR(r) \quad \text{separated sum of resource tuples}$$

where $MR(\mathsf{S}) = ContStack$, $MR(\mathsf{H}) = ContHeap$,
$MR(\mathsf{M}) = Store$, $MR(\mathsf{X}) = Exn$

CONTINUATION ALLOCATION RESOURCES

$$Frame \ni F ::= \mathsf{Bind}\ \langle \lambda^{\rho} x{:}\tau.\ e, E, \tau'\rangle \quad \text{binding activation record}$$

$$\mid \mathsf{Restore}\ \langle r, MR(r)\rangle \quad \text{resource blocking record}$$

$$\mid \mathsf{Drop}\ r \quad \text{resource simulation record}$$

$$MContRes^c \ni C^c \text{ with } empty^c \in MContRes^c$$

$$isEmpty^c \in MContRes^c \to Boolean$$

$$newFrame^c \in Frame \times MContRes^c \to MContRes^c$$

$$topFrame^c \in (isEmpty^c)^{-1}[\{\mathsf{false}\}] \to Frame \times MContRes^c$$

– CONTINUATION STACK

$$ContStack \ni S ::= \mathsf{Halt} \mid \mathsf{Frame}\ \langle F, S\rangle \quad \text{frame stack}$$

$$empty^{\mathsf{S}} = \mathsf{Halt}$$

$$isEmpty^{\mathsf{S}} = [\mathsf{Halt} \mapsto \mathsf{true},\ \mathsf{Frame}\ \langle F, S\rangle \mapsto \mathsf{false}]$$

$$newFrame^{\mathsf{S}}(F, S) = \mathsf{Frame}\ \langle F, S\rangle$$

$$topFrame^{\mathsf{S}}(\mathsf{Frame}\ \langle F, S\rangle) = \langle F, S\rangle$$

– CONTINUATION HEAP

$$ContHeap = ContLoc \times (ContLoc \to Frame \times ContLoc) \times ContLoc$$

where $k \in ContLoc$ continuation locations

$$K \in ContLoc \to Frame \times ContLoc$$

$$empty^{\mathsf{H}} = \langle k, \emptyset, k\rangle \text{ for some } k \in ContLoc$$

$$isEmpty^{\mathsf{H}}(F, \langle k, K, k_e\rangle) = (k = k_e)$$

$$newFrame^{\mathsf{H}}(F, \langle k, K, k_e\rangle) = \langle k', K[k' \mapsto \langle F, k\rangle], k_e\rangle \quad k' \notin Dom(K) \cup \{k_e\}$$

$$topFrame^{\mathsf{H}}(\langle k, K, k_e\rangle) = \langle F, \langle k', K, k_e\rangle\rangle \quad \text{if } K(k) = \langle F, k'\rangle$$

EXCEPTION HANDLER RESOURCE

$$Exn = ContStack \quad (\text{denoted by superscript } \mathsf{X})$$

MUTABLE STORE RESOURCE

$$\ell \in StoreLoc \quad \text{store locations}$$

$$M \in Store = StoreLoc \to MVal$$

with $empty^{\mathsf{M}} = \emptyset$

$$ref^{\mathsf{M}}(w, M) = \langle \ell, M[\ell \mapsto w]\rangle,\ l \notin Dom(M)$$

$$deref^{\mathsf{M}}(\ell, M) = M(\ell)$$

$$update^{\mathsf{M}}(\langle \ell, w\rangle,\ M) = M[\ell \mapsto w],\ l \in Dom(M)$$

Fig. 10. Semantic domains for the instrumented operational semantics of $\mathcal{R}$

$$\gamma(x, E) = {}^{V}E(x)$$

$$\gamma(x[\varepsilon],\ E) = \gamma([\varepsilon/u]v,\ E') \text{ if } {}^{V}E(x) = \mathsf{Closure}\ \langle \Lambda u \le \rho.\ v,\ E'\rangle, \text{ and } \emptyset \vdash_{\varepsilon} \varepsilon \le \rho,$$

$$\gamma(v, E) = \mathsf{Closure}\ \langle v,\ E\rangle \text{ for other } v$$

Fig. 11. Representation of values

and we write $E[x \mapsto (w : \tau)]$ to denote the extension of E which assigns machine value w and type τ to x.

The modularity of the language blocks is supported in our framework by the representation of machine resources Rs as a record (tuple) with a tag $tag\,(Rs) \in Resources$ which yields the set of the corresponding primitive resources. The set of possible values of an individual machine resource corresponding to abstract resource r is denoted by $MR\,(r)$. Thus, assuming a canonical enumeration of resources, a tuple of resources with tag rs is an element of the singleton separated sum $\sum_{rs' \in \{rs\}} \prod_{r \in rs'} MR\,(r)$; we will denote this set by $MR^T\,(rs)$. The semantics make use of two families of total functions,

$$inj^r_{rs-\{r\}} \in MR\,(r) \times MR^T\,(rs-\{r\}) \to MR^T\,(rs \cup \{r\})$$
$$proj^r_{rs \cup \{r\}} \in MR^T\,(rs \cup \{r\}) \to MR\,(r) \times MR^T\,(rs-\{r\}),$$

indexed by a primitive abstract resource r and a tag rs, with the intention that $inj^r_{rs-\{r\}}$ maps $\langle R, Rs\rangle$ to the record containing R and all of Rs, and $proj^r_{rs \cup \{r\}}$ is its inverse. We also use $single^r_{rs} = \pi_1 \circ proj^r_{rs}$ and $drop^r_{rs} = \pi_2 \circ proj^r_{rs}$. The functions inj^r_{rs} and $inj^{r'}_{rs}$ (and similarly $proj$) are commutative for $r \neq r', \{r, r'\} \cap rs = \emptyset$; this commutativity allows the generalization of inj and $proj$ to the functions

$$inj^{rs'}_{rs-rs'} \in MR^T(rs') \times MR^T(rs-rs') \to MR^T(rs \cup rs')$$
$$proj^{rs'}_{rs \cup rs'} \in MR^T(rs \cup rs') \to MR^T(rs') \times MR^T(rs-rs').$$

Using these functions we can define natural liftings of operations linear in an individual resource R (i.e. in whose type R occurs exactly once positively in both the types of domain and codomain, or does not occur in the domain type and the codomain is linear in R) to operations on a record of resources Rs containing R: e.g. for $f^r \in S \times MR\,(r) \to S' \times MR\,(r)$ we define $f^r_{rs}\,(\in) S \times \prod^T_{r \in rs} MR\,(r) \to S' \times \prod^T_{r \in rs} MR\,(r)$ by

$$\begin{aligned} f^r_{rs}\,(s, Rs) = \mathbf{let}\ & \langle R, Rs'\rangle = proj^r_{rs}\,(Rs), \\ & \langle s', R'\rangle = f^r(s, R) \\ \mathbf{in}\ & \langle s', inj^r_{rs-\{r\}}\,(R', Rs')\rangle \end{aligned}$$

where $r \in rs$. With the generalized versions of inj and $proj$ this lifting extends to functions on records of resources as well.

For each primitive resource r there is an initial element $empty^r$ of the algebra of the corresponding machine resource; these elements are used in the simulation of resources (rule (add) in Figure 13).

A generic continuation machine resource C^c (corresponding to abstract control resource c) is described by the four stack operations $empty^c$, $isEmpty^c$, $newFrame^c$, and $topFrame^c$, where $topFrame^c$ is the inverse of $newFrame^c$ on non-empty C^c. We will use the more concise notation $F ::^c_{rs} Rs$ both for $newFrame^c_{rs}\,(F, Rs)$ and as a pattern standing for Rs' when $c \in rs$, $isEmpty^c_{rs}\,(Rs') = \mathsf{false}$, and $topFrame^c_{rs}\,(Rs') = \langle F, Rs\rangle$. A section $(F ::^c_{rs})$ stands for the function mapping Rs to $F ::^c_{rs} Rs$.

The stack-based implementation only provides the minimal functionality; the heap-based implementation can be used for non-sequential access as well. Exception handling uses a separate continuation stack.

Activation frames include a variant of the standard binding frames [3] to record the continuation completing a **let**-binding $\textbf{let}^{\langle a,rs\rangle}\ x : \tau' \leftarrow e'\ \textbf{in}\ e$ after evaluating e'. The binding frame is of the form $\mathsf{Bind}\ \langle \lambda^{\langle a,rs\rangle}\, x : \tau.\ e,\ E,\ \tau' \rangle$, where τ' is the type of e in environment ${}^{T}E$ extended with $x : \tau$, rs is the set of resources available for the evaluation of the **let**, and a is the allocation resource for the evaluation of e.

In addition there are two kinds of resource management continuation frames. A Restore frame indicates that a machine resource has been saved and removed from the current; a Drop frame signals that a dummy resource has been created and provided to code which only uses the it locally, or not at all (but has been compiled to expect it). The names of these frames suggest the operations that must be done to restore the status after the evaluation of the current term has completed, i.e. when the frame is encountered during unwinding the continuation.

The relation of computation $\mapsto_1$ on configurations is the union of the transition rules shown in Figures 12 and 13); the relation of computation is the reflexive transitive closure of $\mapsto_1$. The transition rules are grouped in classes based on the feature they implement as follows.

Building block	Rules	Figure
environment	(app), (let), (bind)	12
store	(ref), (deref), (update)	12
first-class continuations	(callcc), (throw)	13
exceptions	(handle), (raise)	13
resource management	(add), (remove), (redirect) (nop), (restore), (drop)	13

Notable among the transitions are those for resource management – they save a resource for future use and remove it from the currently available set, create a dummy resource, or select a different continuation. The interaction between resource management and exception handling is non-trivial because resources must be restored when exiting a **use**-region in any way. For that reason first-class continuations are restricted to accept only the resource set available at point of capture; unlike them, exception handlers are allocated on a stack, thus it is possible to allow exceptions not to be tied to specific resources by intercepting them and restoring the resources.

Our implementation of exceptions is more realistic than the typical [23] which propagates exception packets through all enclosing terms whose evaluations is pending. The exception handler forms a stack parallel to the continuation stack (or heap). Raising an exception (rule (raise)) creates an exception packet which is processed like a value with continuation on the exception handler stack; the bindings on this stack, created by **handle**, restore the default continuation. This scheme has on overhead if exceptions are not used, and overhead linear in number of catching handlers when an exception is thrown.

3.3 Soundness of the Type System

To prove soundness of the type system we extend it with rules for machine configurations, and prove that this extension has the subject reduction and progress properties.

ENVIRONMENT

(app) $\langle @\ x_1\ x_2,\ E,\ a,\ Rs,\ \varepsilon,\ \tau\rangle \mapsto_1 \langle e',\ E'[x' \mapsto E(x_2)],\ a,\ Rs,\ \varepsilon,\ \tau\rangle$

where $\mathsf{Closure}\ \langle \lambda^{\langle a,rs\rangle}\ x' : \tau'.\ e', E'\rangle = {}^{V}E(x_1)$

$\tau' = {}^{T}E(x_2)$

(let) $\langle \mathbf{let}^{\langle a,rs\rangle}\ x : \tau' \leftarrow e'\ \mathbf{in}\ e,\ E,\ a,\ Rs,\ \varepsilon,\ \tau\rangle \mapsto_1$
$\langle e',\ E,\ a,\ \mathsf{Bind}\ \langle \lambda^{\langle a,rs\rangle}\ x : \tau'.\ e, E|_{FV(e)-\{x\}}, \tau\rangle ::^{a}_{rs} Rs,\ \varepsilon,\ \tau'\rangle$

(bind) $\langle \lfloor v \rfloor^{\langle c,rs\rangle},\ E,\ c,\ \mathsf{Bind}\ \langle \lambda^{\langle a,rs\rangle}\ x : \tau.\ e', E', \tau'\rangle ::^{c}_{rs} Rs',\ \varepsilon,\ \tau\rangle \mapsto_1$
$\langle e',\ E'[x \mapsto (\gamma(v, E) : \tau)],\ a,\ Rs',\ \varepsilon,\ \tau'\rangle$

STORE (rules valid when $\mathsf{M} \in rs$)

(ref) $\langle \mathbf{ref}^{\langle a,rs\rangle}\ x,\ E,\ a,\ Rs,\ \varepsilon,\ \tau\rangle \mapsto_1$
$\langle \lfloor x' \rfloor^{\langle a,rs\rangle},\ E[x' \mapsto (\mathsf{Loc}\ \ell : \tau)],\ a,\ Rs',\ \varepsilon \cup \{\mathsf{store}\},\ \tau\rangle$

where $\tau = \mathsf{ref}[{}^{T}E(x)]$

$\langle \ell, Rs'\rangle = ref^{\mathsf{M}}_{rs}\ ({}^{V}E(x),\ Rs)$

$x' \notin Dom\,(E)$

(deref) $\langle !^{\langle a,rs\rangle}\ x,\ E,\ a,\ Rs,\ \varepsilon,\ \tau\rangle \mapsto_1$
$\langle \lfloor x' \rfloor^{\langle a,rs\rangle},\ E[x' \mapsto (deref^{\mathsf{M}}_{rs}\ (\ell, Rs) : \tau)],\ a,\ Rs,\ \varepsilon \cup \{\mathsf{store}\},\ \tau\rangle$

where $(\mathsf{Loc}\ \ell : \mathsf{ref}[\tau]) = E(x)$

$x' \notin Dom\,(E)$

(update) $\langle x :=^{\langle a,rs\rangle} x',\ E,\ a,\ Rs,\ \varepsilon,\ \mathsf{unit}\rangle \mapsto_1$
$\langle \lfloor * \rfloor^{\langle a,rs\rangle},\ E,\ a,\ update^{\mathsf{M}}_{rs}\ (\langle \ell,\ {}^{V}E(x')\rangle, Rs),\ \varepsilon \cup \{\mathsf{store}\},\ \mathsf{unit}\rangle$

where $(\mathsf{Loc}\ \ell : \tau) = E(x)$

$\tau = \mathsf{ref}[{}^{T}E(x')]$

where $rs = tag\,(Rs),\ a \in rs$

Fig. 12. Instrumented transition rules, part 1

The interesting aspect of assigning a type to a configuration in this system is that some of the values in the environment may refer to resources which are currently inaccessible (blocked by an enclosing **use**), which complicates the notion of type correctness. The details of the proof are omitted for space considerations.

The progress and subject reduction properties are combined in the following lemma.

Lemma 1 (Safety). *If $C = \langle e,\ E,\ c,\ Rs,\ \varepsilon,\ \tau\rangle$ and $\emptyset \vdash_C C : \tau_0; \varepsilon_0$, then either $e = \lfloor v \rfloor^{\langle c,rs\rangle}$ for some value v such that $\emptyset; {}^{T}E \vdash_v v : \tau$, and $isEmpty^{c}_{rs}\ (Rs) = \mathsf{true}$, or there exists a configuration C' such that $C \mapsto_1 C'$ and $\emptyset \vdash_C C' : \tau_0; \varepsilon_0$.*

(Note that the case of a value in an empty continuation covers both normal termination of the program and the case of an unhandled exception propagated to the top.)

As a corollary we obtain soundness of the system.

Theorem 2 (Soundness). *If $C = \langle e,\ E,\ c,\ Rs,\ \varepsilon,\ \tau\rangle$ and $\emptyset \vdash_C C : \tau_0; \varepsilon_0$, then C computes to a value, or to an exception packet, or its computation diverges.*

FIRST-CLASS CONTINUATIONS (rules valid when $\mathsf{H} \in rs$)

(callcc) $\langle \mathbf{callcc}^{\langle \mathsf{H}, rs\rangle}\ x,\ E,\ Rs,\ \mathsf{H},\ \varepsilon,\ \tau\rangle \mapsto_1$
$\langle e',\ E'[x' \mapsto (\mathsf{Cont}\ k : \mathsf{cont}^{\langle \mathsf{H}, rs\rangle}[\tau])],\ \mathsf{H},\ Rs',\ \varepsilon \cup \{\mathsf{callcc}\},\ \tau\rangle$
where $\langle k, K, k_e\rangle = single^{\mathsf{H}}_{rs}\,(Rs)$
$Rs' = \mathbf{if}\ \mathsf{X} \notin rs\ \mathbf{then}\ Rs\ \mathbf{else}\ restoreExn\ (\mathsf{H})(Rs)$
${}^{V}E(x) = \mathsf{Closure}\ \langle \lambda^{\langle \mathsf{H}, rs\rangle}\ x' : \mathsf{cont}^{\langle \mathsf{H}, rs\rangle}[\tau].\ e', E'\rangle$

(throw) $\langle \mathbf{throw}^{\langle \mathsf{H}, rs\rangle}[\tau]\ x_1\ x_2,\ E,\ a,\ Rs,\ \varepsilon,\ \tau\rangle \mapsto_1$
$\langle \lfloor x_2 \rfloor^{\langle \mathsf{H}, rs\rangle},\ E,\ \mathsf{H},\ inj^{\mathsf{H}}_{rs-\{\mathsf{H}\}}\,(\langle k_1, K, k_e\rangle,\ Rs'),\ \varepsilon,\ \tau''\rangle$
where $(\mathsf{Cont}\ k_1 : \mathsf{cont}^{\langle \mathsf{H}, rs\rangle}[\tau'']) = E(x_1)$
$\langle \langle k, K, k_e\rangle,\ Rs'\rangle = proj^{\mathsf{H}}_{rs}\,(Rs)$

EXCEPTIONS (rules valid when $\mathsf{X} \in rs$)

(handle) $\langle e\ \mathbf{handle}^{\langle a, rs\rangle}\ x : \mathsf{exn}.\ e',\ E,\ a,\ Rs,\ \varepsilon,\ \tau\rangle \mapsto_1 \langle e,\ E,\ a,\ Rs',\ \varepsilon \cup \{\mathsf{exception}\},\ \tau\rangle$
where $Rs' = \left(\begin{array}{l} restoreCont\ (a, Rs) \circ \\ (\mathsf{Bind}\ \langle \lambda^{\langle a, rs\rangle}\ x : \mathsf{exn}.\ e', E|_{FV\,(e')-\{x\}}, \tau\rangle ::^{\mathsf{X}}_{rs}) \circ \\ restoreExn\ (a) \end{array}\right)(Rs)$

(raise) $\langle \mathbf{raise}^{\langle a, rs\rangle}[\tau]\ x,\ E,\ a,\ Rs,\ \varepsilon,\ \tau\rangle \mapsto_1 \langle \lfloor x \rfloor^{\langle \mathsf{X}, rs\rangle},\ E,\ \mathsf{X},\ Rs,\ \varepsilon \cup \{\mathsf{exception}\},\ \mathsf{exn}\rangle$

RESOURCE MANAGEMENT

(add) $\langle \mathbf{use}^{\langle a, rs\rangle}\ (\langle a, rs'\rangle)\,e,\ E,\ a,\ Rs,\ \varepsilon,\ \tau\rangle \mapsto_1$
$\langle \mathbf{use}^{\langle a, rs\cup\{r\}\rangle}\ (\langle a, rs'\rangle)\,e,\ E,\ a,\ inj^{r}_{rs}\,(empty^{r},\ \mathsf{Drop}\ r ::^{a}_{rs} Rs'),\ \varepsilon,\ \tau\rangle$
where $r \in rs' - rs$
$Rs' = \mathbf{if}\ \mathsf{X} \notin rs\ \mathbf{then}\ Rs\ \mathbf{else}\ \mathsf{Drop}\ r ::^{\mathsf{X}}_{rs} restoreExn\ (a)(Rs)$

(block) $\langle \mathbf{use}^{\langle a, rs\rangle}\ (\langle a, rs'\rangle)\,e,\ E,\ a,\ Rs,\ \varepsilon,\ \tau\rangle \mapsto_1$
$\langle \mathbf{use}^{\langle a, rs-\{r\}\rangle}\ (\langle a, rs'\rangle)\,e,\ E,\ a,\ \mathsf{Restore}\ \langle r, R\rangle ::^{a}_{rs} Rs'',\ \varepsilon,\ \tau\rangle$
where $r \in rs - rs' - \{a\}$
$Rs'' = \mathbf{if}\ \mathsf{X} \notin rs'\ \mathbf{then}\ Rs\ \mathbf{else}\ \mathsf{Restore}\ \langle r, R\rangle ::^{\mathsf{X}}_{rs} restoreExn\ (a)(Rs)$
$\langle R, Rs'\rangle = proj^{r}_{rs}\,(Rs)$

(redirect) $\langle \mathbf{use}^{\langle a, rs\rangle}\ (\langle a', rs'\rangle)\,e,\ E,\ a,\ Rs,\ \varepsilon,\ \tau\rangle \mapsto_1$
$\langle \mathbf{use}^{\langle a', rs\rangle}\ (\langle a', rs'\rangle)\,e,\ E,\ a',\ \mathsf{Bind}\ \langle \lambda^{\langle a, rs\rangle}\ x : \tau.\ \lfloor x \rfloor^{\langle a, rs\rangle},\ E,\ \tau\rangle ::^{a'}_{rs} Rs,\ \varepsilon,\ \tau\rangle$

(nop) $\langle \mathbf{use}^{\langle a, rs\rangle}\ (\langle a, rs\rangle)\,e,\ E,\ a,\ Rs,\ \varepsilon,\ \tau\rangle \mapsto_1 \langle e,\ E,\ a,\ Rs,\ \varepsilon,\ \tau\rangle$

(restore) $\langle \lfloor v \rfloor^{\langle c, rs\rangle},\ E,\ c,\ \mathsf{Restore}\ \langle r, R\rangle ::^{c}_{rs} Rs',\ \varepsilon,\ \tau\rangle \mapsto_1$
$\langle \lfloor v \rfloor^{\langle c, rs\rangle},\ E,\ c,\ inj^{r}_{rs}\,(R,\ Rs'),\ \varepsilon,\ \tau\rangle$
where $c \in rs,\ r \notin rs$

(drop) $\langle \lfloor v \rfloor^{\langle c, rs\rangle},\ E,\ c,\ \mathsf{Drop}\ r ::^{c}_{rs} Rs',\ \varepsilon,\ \tau\rangle \mapsto_1 \langle \lfloor v \rfloor^{\langle c, rs\rangle},\ E,\ c,\ drop^{r}_{rs}\,(Rs'),\ \varepsilon,\ \tau\rangle$
where $c \in rs,\ r \in rs - \{c\}$

where
$rs = tag\ (Rs)$
$a \in rs$
$restoreExn\ (a)(Rs) = replaceResource\ (a, \mathsf{X}, Rs)(Rs)$
$restoreCont\ (a, Rs') = replaceResource\ (\mathsf{X}, a, Rs')$
$replaceResource\ (c, r, Rs') = (\mathsf{Drop}\ r ::^{c}_{rs}) \circ (\mathsf{Restore}\ \langle r, single^{r}_{rs}\,(Rs')\rangle ::^{c}_{rs})$
where $tag\ (Rs') = rs,\ \{c, r\} \subseteq rs$

Fig. 13. Instrumented transition rules, part 2

4 Related Work

Interoperability is a primary concern of component-based models such as CORBA [12] and Microsoft's COM [15, 13]. Safety in these systems is in conflict with the performance requirements, and even sandboxing may fail to provide correct semantics when the effect systems of the communicating languages go beyond the value/store interface. In comparison our system allows flexible and efficient interoperation between languages with different resources, and ensures safety by exposing the resource and effect requirements in the types of the components.

Foreign function call interfaces [6, 14] are related in purpose but are designed under the constraint to be compatible with legacy code which is often unsafe. Solutions to this problem have major impact on interoperability today. We do not attempt to solve compatibility issues in the system presented in this paper. Our design is for interoperability between safe components with language-independent interfaces, aimed to satisfy high performance requirements when running in shared address space. We emphasize building a *safe*, *efficient*, and *robust* interface across multiple HOT languages.

The present work extends our earlier results [17] on a type system with effect and resource control for continuation allocation. While the state resource is essentially independent of the rest, the interactions of the exception resource with the continuation resources are non-trivial.

Although we do not present the semantics in terms of monads, the idea to use both resources and effects to describe a function's interoperability was inspired by recent work on monad-based interactions and modular interpreters [21, 9, 20, 10], and Wadler's work on the relationship between monads and effects [22]. Monads, viewed as compositions of basic monad transformers [10], can be used to represent sets of resources. The transition rules creating a binding activation frame and binding a variable to a value (Figure 12), given a set of resources, provide the semantics of the `'bind'` and `unit` of the monad. Resource-specific primitives are interpreted by lifting the operations of the corresponding monad through the monad transformers enclosing it in the composition.

What makes our approach different is that our system keeps track of effects, which allows us to determine which components of a monad transformer composition are being used only trivially (i.e. only their `'bind'` and `unit` are invoked) and therefore can be eliminated or simulated. Furthermore, we wish to extend and reduce the set of resources in a commutative way, and for that purpose we represent the result of transformer compositions "horizontally" – the corresponding resources are collected in one component of the abstract machine configuration. Thus if the set of resources required by term e corresponds to the monad $M = (T_1 \circ \ldots \circ T_n)Id$, extending it via $\mathbf{use}^{\rho}\ (e)$ is, in general, not expressed as an application of a monad transformer T to M, but as the use of a monad morphism [2] to embed values of M into a monad isomorphic to a composition of $T_1, \ldots, T_n, T$ in a canonical order. Proving the equivalence of a monads defined as compositions with their horizontal representations meets the technical complexity of constructing morphisms between them. We have opted instead for operational semantics which gives a more direct correspondence with an implementation. In our semantics the complexity of lifting monads through monad transformers is reflected

in the interaction between various resources, e.g. in transition rules (callcc), (add), and (block) (Figure 13).

Closely related to our work is the research on effect systems [4, 7, 8, 18, 19]. The effects in our system are used for verifying interoperability constraints; in this context the novel bounded effect quantification is introduced as a form of effect polymorphism under resource restrictions which allows us to take advantage of the effect-resource relationship to support advanced compilation strategies. The effect inference suggested in the typing rules in Figure 8 is conservative and there is considerable room for improvement borrowing from prior work by e.g. finer separation of effects and adopting region inference or for determining their localization; however it is still not clear to what extent this improvement will materialize in practice – for instance the exception effects can be easily localized to a handler, but due to the typical extensibility of the exception type most handlers re-raise exceptions. We intend to test these variations in a prototype implementation under development within the FLINT system.

Acknowledgment

We thank the anonymous referees for suggestions on improving the presentation.

References

[1] William D Clinger, Anne H Hartheimer, and Eric M Ost. Implementation strategies for continuations. In *ACM Conference on Lisp and Functional Programming*, pages 124–131, New York, June 1988. ACM Press.

[2] Andrzej Filinski. *Controlling Effects*. PhD thesis, CMU, 1996.

[3] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 237–247, New York, June 1993. ACM Press.

[4] David K. Gifford, Pierre Jouvelot, John M. Lucassen, and Mark A. Sheldon. FX-87 reference manual. Technical Report MIT/LCS/TR-407, M.I.T. Laboratory for Computer Science, September 1987.

[5] Robert Hieb, R. Kent Dybvig, and Carl Bruggeman. Representing control in the presence of first-class continuations. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 66–77, New York, 1990. ACM Press.

[6] Lorenz Huelsbergen. A portable C interface for Standard ML of New Jersey. Technical memorandum, AT&T Bell Laboratories, Murray Hill, NJ, January 1996.

[7] Pierre Jouvelot and David K. Gifford. Reasoning about continuations with control effects. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 218–226. ACM Press, 1989.

[8] Pierre Jouvelot and David K. Gifford. Algebraic reconstruction of types and effects. In *18th Annual ACM Symposium on Principles of Programming Languages*, pages 303–310, New York, Jan 1991. ACM Press.

[9] John Launchbury and Simon Peyton Jones. Lazy functional state threads. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 24–35, New York, June 1994. ACM Press.

[10] Sheng Liang, Paul Hudak, and Mark Jones. Monad transformers and modular interpreters. In *22nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 333–343. ACM Press, 1995.

[11] Greg Nelson, editor. *Systems programming with Modula-3.* Prentice Hall, Englewood Cliffs, NJ, 1991.

[12] Object Management Group (OMG). The Common Object Request Broker: Architecture and specifications (CORBA). Revision 1.2., Object Management Group (OMG), Framingham, MA, December 1993.

[13] Simon Peyton Jones, Eric Meijer, and Daan Leijen. Scripting COM components in Haskell. Available at http://www.dcs.gla.ac.uk/~simonpj/com.ps.gz, 1997.

[14] Simon Peyton Jones, Thomas Nordin, and Alastair Reid. Green card: A foreign-language interface for Haskell. Available at http://www.dcs.gla.ac.uk/fp/authors/Simon_Peyton_Jones/green-card-1.ps.gz, 1997.

[15] Dale Rogerson. *Inside COM: Microsoft's Component Object Model*. Microsoft Press, 1997.

[16] Zhong Shao. Typed common intermediate format. In *USENIX Conference on Domain Specific Languages*, pages 89–102, October 1997.

[17] Zhong Shao and Valery Trifonov. Type-directed continuation allocation. In *2nd International Workshop on Types in Compilation*, volume 1473 of *LNCS*, pages 116–135, Berlin, 1998. Springer.

[18] Jean-Pierre Talpin and Pierre Jouvelot. Polymorphic type, region, and effect inference. *Journal of Functional Programming*, 2(3), 1992.

[19] Jean-Pierre Talpin and Pierre Jouvelot. The type and effect discipline. *Information and Computation*, 111(2):245–296, June 1994.

[20] Philip Wadler. The essence of functional programming (invited talk). In *19th Annual ACM Symposium on Principles of Programming Languages*, New York, Jan 1992. ACM Press.

[21] Philip Wadler. How to declare an imperative (invited talk). In *International Logic Programming Symposium*, Portland, Oregon, December 1995. MIT Press.

[22] Philip Wadler. The marriage of effects and monads. In *ACM SIGPLAN International Conference on Functional Programming*, pages 63–74. ACM Press, 1998.

[23] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, November 1994.